LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-427633

I/O Performance of a Large-Scale, Interpreter-Driven Laser-Plasma Interaction Code

T. Gamblin, S. H. Langer, B. Still, R. Hedges, M. Schulz, B. R. de Supinski

April 13, 2010

Super Computing Conference
New Orleans, LA, United States
November 13, 2010 through November 19, 2010

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

I/O Performance of a Large-Scale, Interpreter-Driven Laser-Plasma Interaction Code

Todd Gamblin, Steven H. Langer, Bert Still, Richard Hedges, Martin Schulz, and Bronis R. de Supinski
Lawrence Livermore National Laboratory, 7000 East Avenue, Livermore CA 94550
{tgamblin, langer-steve, bertstill, richard-hedges, schulzm, bronis}@llnl.gov

***Abstract*—I/O to compute ratios of large-scale supercomputers are falling and I/O systems are becoming dauntingly complex. Applications may communicate with parallel filesystems through special-purpose I/O nodes and large networks of file servers. Peak bandwidth on such systems depends on the hardware, the filesystem, and an application's I/O patterns. Application developers must understand the I/O system's architecture in detail to obtain good performance. Many large simulations are interpreter-driven, which may increase the difficulty measuring low-level I/O operations. Obtaining detailed I/O measurements may be a scalable I/O problem of its own for simulations that run on hundreds of thousands of cores,**

We have developed a tool to scalably measure and analyze per-process I/O in interpreted environments. We describe our experiences applying our tool to understand and tune the performance of pF3D, an interpreter-driven laser-plasma interaction code that regularly runs on over 200,000 cores.

I. Introduction

This paper discusses techniques to measure and optimize the I/O performance of pF3D. pF3D is a multi-physics code that carries out time-depemndent simulations of the interaction between a laser beam and a plasma in experiments being carried out at the National Ignition Facility (NIF). pF3D simulations currently use over 55 billion zones and work is underway on a multiple beam simulation capability that will significantly increase the number of zones. Full beam simulations run for several weeks on massively parallel computers. pF3D has run on 192K BlueGene/L cores and 144K BlueGene/P cores with excellent scaling.

A checkpoint of the current state of a pF3D simulation may exceed 20 TB and pF3D also saves frequent (smaller) diagnostic data sets. Either of these dump processes can consume more time than the computation unless care is taken in writing the I/O package.

The ratio of I/O bandwidth to compute performance will drop significantly in future large supercomputers. For example, LLNL's current 0.5 PFLOP/s Dawn system has a disk bandwidth to peak floating point performance ratio of 384 GB/s per PFLOP/s while the 20 PFLOP/s Sequoia system (targeted for production use in 2012) will have a ratio of 25.6. That will significantly increase the I/O challenges faced by a code like pF3D.

Computational steering has become popular in large parallel simulations. The disk I/O, graphics, and code steering are typically handled by an interpreter (e.g. Yorick or Python) and the computational work is still done in a compiled language like Fortran, C, or C++. Python may be used for the same purpose. Using an interpreter for I/O improves programmer and user productivity and makes it easier to try new I/O strategies. The additional abstractions used by the interpreter can hurt I/O performance if the programmer is not aware of their implications.

It is clear that pF3D will face serious challenges in achieving the desired I/O performance on future systems. One of the challenges in trying to improve the I/O performance of a massively parallel code is the difficulty of gathering the performance data required to evaluate alternate I/O strategies. Caution must be exercised to avoid having the collection of the performance data become a bigger challenge than the I/O itself.

In this paper we employ a data collection technique that greatly reduces the amount of performance data that needs to be stored by collapsing data into equivalence classes.

II. pF3D Overview

pF3D is a multi-physics code that carries out time-dependent simulations of the interaction between a laser beam and a plasma in experiments being carried out at the National Ignition Facility (NIF). pF3D solves wave equations for the laser light and two kinds of backscattered light. The light waves are coupled together through interactions with electron plasma waves and ion acoustic waves.

pF3D has been implemented as a set of compiled C routines connected to the Yorick interpreter (D. H. Munro, Computers In Physics **9** (6), 609; yorick.sf.net). Yorick handles I/O, graphics, input decks, and code steering during a run. Yorick has a package called mpy that permits interpreted functions to pass messages between tasks using MPI. mpy is used to coordinate I/O and to implement the time step loop.

The equations are solved on a 3D Cartesian grid. Parallelization is carried out by splitting the grid into equal chunks with decomposition in all three dimensions. A NIF beam is comprised of many small bright spots ("speckles"). An accurate solution to the equations can be obtained only by resolving the speckle structure, which requires the zones to be at most a couple of wavelengths across. The laser wavelength is roughly 0.3 μm and the plasma volume that needs to be simulated is several mm across. The net result is that the simulation of the interactions between the light waves and the plasma over the full volume of one NIF beam requires over 55 billion zones. Efforts are currently underway to simulate

the interaction between two NIF beams and those may require hundreds of billions of zones.

The wave equations are solved in the paraxial approximation which assumes all light waves are traveling nearly in the z-direction (the laser propagation direction). The wave propagation and coupling are solved using Fourier transforms in xy-planes. The Fourier transforms require message passing across the full extent of a transverse plane. A significant fraction of the run time is spent passing messages. The message passing makes pF3D a tightly coupled, non-local code.

pF3D also solves the hydrodynamic equations. The light equations are sub-cycled because light moves much faster than sound waves. In a typical run, the light sub-cycle is the time for light to cross one zone and the hydrodynamic equations are solved every 50 sub-cycles.

Full beam simulations run for several weeks on massively parallel computers. pF3D has run on 192K BlueGene/L cores and 144K BlueGene/P cores with excellent scaling. Work is currently underway to prepare pF3D for million way parallelism on the upcoming Sequoia system.

pF3D has many variables in each zone, so the state of the simulation occupies a large fraction of the computer's memory. A large pF3D currently writes 30 TB checkpoint restart sets as a fault tolerance strategy. Write rates should be high enough that pF3D spends most of its time computing. The performance of current parallel file systems is not high enough to permit us to write checkpoints to disk as often as we would like to achieve high efficiency in the face of hardware and software errors. pF3D uses checkpoints to memory or solid state disk on X86 clusters to achieve the desired efficiency without over burdening the file system (*Scalable I/O Systems via Node-Local Storage: Approaching 1 TB/sec File I/O*; Moody and Bronovetsky, SC08). We plan to extend this technique to BlueGene systems in the future.

The scientific results of a simulation are extracted by examining how the state variables vary in space and time. The data sets saved for volume visualization are compressed, but still consume 10-20 TB by the end of current large runs. pF3D also saves information on a few key planes (e.g. the entrance and exit planes) very frequently during the run. This data (called "spec dumps") can be used, amongst other things, to generate an estimate of the spectrum of the backscattered light for comparison to data collected during NIF experiments. These dumps are small compared to checkpoint or visualization dumps, but they occur so frequently (every couple of light sub-cycles) that pF3D still needs to write them at high data rates. Obtaining high I/O rates for small dumps is a serious challenge on current systems.

Parallel file systems work best if the hardware sees large, contiguous data transfers. Large data blocks permits striping across multiple RAID sets and enhances performance for a specific block transfer. A block from pF3D does not need to be large enough to stripe across all RAID sets since there are many files being written simultaneously.

Yorick has an internal cache block system. The default size of a cache block is 64 kB, but the user can increase the size. Yorick accumulates data until a cache block is full, then it issues a write() call to start the data on its way to disk. The requirement for good I/O performance is thus that the cache block be large enough, not that individual writes in the Yorick interpreted language involve large numbers of bytes.

The operating system on the compute node, the operating system on the I/O node, or the computers that are physically connected to the RAID sets may all choose to provide further buffering. This buffering may lead to large writes to the physical disk even if the Yorick cache block is "too small". The complexity of this buffering system makes it hard to assess whether an I/O scheme is using appropriate sizes for its write requests.

There is a good chance that a seek to a location other than the current byte in the file will force a flush of all layers of buffer to the physical disk. A file format that alternates between writing "symbol table" information in the middle of the file and data at the end file will lead to lots of small writes to physical disk unless the individual items that are written are large.

A good first step towards high I/O performance is thus to ensure that writes progress sequentially from byte zero to the end of the file rather than seeking back and forth through the file. This allows buffering to mask the size of individual writes within the program. Only after writes are sequential does the programmer need to worry about the size of individual writes.

Achieving high performance on large parallel file systems is not easy. A dump file set that is perfectly balanced at an algorithmic level may see significant imbalances induced by slow performance from individual RAID sets (e.g. a RAID set that is rebuilding a parity disk). There are also important differences between the popular parallel file systems. A programmer needs to handle the interactions between messages moving across the interconnect, I/O nodes moving bytes to the file system front end nodes, block sizes of the underlying disk stripes, etc. Metadata operations are expensive for large file sets and can have significantly different characteristics on different parallel file systems. Performance may also be significantly affected by other jobs reading and writing to the same parallel file system. The OSTs to which data are assigned in a Lustre file system are at best loosely under the programmer's control, but they may have a significant impact on performance.

III. pF3D I/O Schemes

All I/O in pF3D is written in the Yorick interpreted language. Files are written using normal Posix I/O, not MPI I/O. High performance can be achieved if the code developer understands the characteristics of the parallel file system and crafts a suitable I/O scheme.

pF3D is a follow-on to a scalar Fortran code called F3D that ran on Cray Y/MP and J-90 systems. The original target for pF3D was a few hundred processors. At this scale, writing an independent checkpoint file to disk from each process and passing the information for the spec dump back to the MPI rank zero process for output to a single spec file worked well.

Visualization dumps were added when thousand processor systems became available and were written in a file per process mode.

When ten thousand processor systems became available, it was no longer viable to gather all the information needed for spec dumps back to the rank zero process both because it was an Amdahl's Law performance bottleneck and because rank zero might not have enough spare memory on BlueGene systems. pF3D switched to a scheme where a subset of the processors are assigned as I/O group leaders that are responsible for all disk I/O for their group members. This scheme still uses Posix I/O since only one process writes to a given file. mpy is used to pass information to the group leader and to coordinate the activities of the group members. The number of processes per I/O group is a tunable parameter. I/O groups tend to have more members on BlueGene systems since they have slower processors and less memory per core than our X86 systems.

Several variants of this dump scheme have been written over the last few years. They have improved performance, but more rapid progress could have been made if we had accurate and detailed performance measurements from large runs. This paper discusses a data collection scheme that gives the pF3D developers this information.

To simplify the presentation of the performance measurements, we will name several I/O strategies.

a) One File: This is the scheme where the spatially decomposed information required for the spec dumps is sent to the rank zero process and assembled into full planes there. Planes currently have roughly 2000x3000 cells and a typical run saves several variables on 8-10 planes. This scheme has the virtue that scientists can easily examine the re-assembled planes, but it has nothing to recommend it from the performance point of view. This scheme has never been used for visualization dumps.

b) Multi-Message: This scheme writes one file per I/O group. The group members send their data to their group leader using one mpy message per variable. The group leader writes the variables to the file with a name reflecting both the domain and the variable name.

In the case of visualization dumps, all processes send the same amount of data back to their group leader and the amount of data per message is large enough that interconnect latency is not an issue on current machines.

The planes of data for spec dumps intersect some domains and not others. As a result, group members send different amounts of data back to their group leader. The amount of data per message is smaller than for visualization dumps and interconnect latency might be a performance issue.

c) Single Message: Yorick recently added the capability to write in-memory files. In this I/O scheme, each process writes all of its data to a file in its own memory. The bytes making up this file are sent as a single message to the group leader which then writes the data to the group's file. In the case of spec dumps, the data is written as a single variable per domain. This results in many logical files within a single physical file. Post-processing software opens one logical file at a time. Visualization dumps may either write a logical file per domain (like spec dumps) or write each individual variable out with the same names as the multi-message scheme for compatibility with existing visualization software.

d) BlueGene I/O Mapping: There are variants of these schemes that deal with the special characteristics of the BlueGene systems. BlueGene L and P systems have sets of compute nodes physically connected to an I/O node (psets). I/O requests are function shipped from a compute node to its I/O node. If I/O groups are assigned as contiguous chunks of MPI ranks, the group leaders are not evenly divided between the I/O nodes. For some choices of group size, there are I/O nodes that don't have any I/O group leaders amongst their pset. This results in reduced I/O performance. To deal with this feature, pF3D now has "uniform I/O groups" and "BlueGene aware I/O groups". The BlueGene aware groups assign an equal number of group leaders from each pset. The division of the group leaders amongst the psets must be independent of the assignment of MPI ranks to physical locations on the interconnect torus because complex mappings may be required to maximize the message passing in the Fourier transforms.

e) Startup I/O: There is one other type of I/O that needs to be optimized for very large clusters. That is the reading of yorick's initialization files and the initial physical conditions for the simulation. Reading these files had never been a serious performance concern because they are read once at the start of the run. All processes read exactly the same files, so caching on the file server always worked well. LLNL recently turned off file caching on one of its Lustre file systems to increase performance for streaming I/O. That dramatically slowed the performance for reading the initialization files. A temporary work around has been to move the input files to an "NFS toaster" which still has caching, but that might not scale well on million core systems. A new version of mpy has recently become available, and it has the ability to broadcast load files. This approach should allow initialization to easily scale to over a million cores and is currently undergoing testing.

IV. Scalably Clustering Performance Data

pF3D I/O schemes involve copious network traffic and require a large amount of I/O bandwidth. To understand the causes of performance problems, it is necessary to collect performance data from many processes in the system, and to assess how efficiently they output data, either through MPI or through the I/O system on a large cluster.

In pF3D, I/O performance measurement is challenging for two reasons. First, pF3D may use upwards of 200,000 processes, and we need a scalable mechanism to gather performance data from all of them. As mentioned, I/O to compute ratios are decreasing rapidly in large systems, and pF3D already makes heavy use o the host machine's I/O system. We cannot afford to ship large volumes of performance data out of the compute partition, because it may interfere with the pF3D traffic we are trying to measure. Second, pF3D runs in the interpreted Yorick environment, and detailed I/O

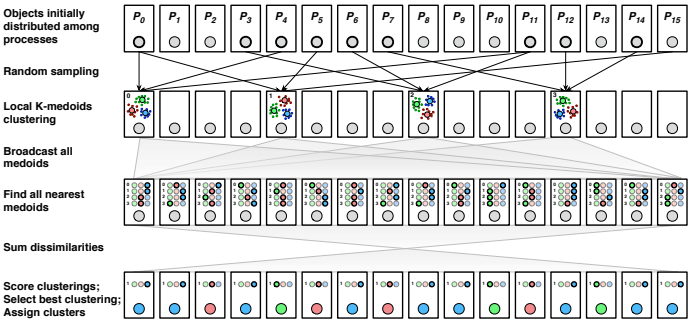

Fig. 1: The CAPEK clustering algorithm.

performance information is not available in the interpreted environment. Therefore, we must provide low-level instrumentation to intercept key I/O and communication calls underneath Yorick to understand the impact of different dump schemes on the system.

To solve these problems and to enable detailed I/O analysis at scale, we have developed the CAPEK clustering algorithm [**?**]. CAPEK is designed for clustering massively distributed performance data from large numbers of processors, and we have shown its run-time to scale sub-linearly to hundreds of thousands of cores. The advantage of this approach is that it enables us to consider performance data as a set of equivalence classes rather than as a large number of independent data points. Each cluster in the output of the algorithm represents a logical group of similar performance data points, rather than the exhaustive set. This is useful both for gaining insight to the data, in that it is easier for humans to understand, and for data reduction, as a set of representatives from equivalence classes is much smaller than the full, unprocessed set of performance data.

CAPEK achieves this scalability through sampling. We give a brief overview of the algorithm here to provide background for our analysis. Figure **??** shows a high-level overview of CAPEK. CAPEK takes as input a set of distributed performance data elements and a distance metric defined on those elements. The distance metric is used to determine which elements are similar to each other and thus which elements should be in the same cluster.

Initially, performance data (shown as gray circles) is distributed across all processes (represented by boxes). CAPEK then selects a number of random subsets of the full set of processes and aggregates them to a small number of worker nodes. The workers perform sequential k-medoids [**?**] clustering on their subset of the full data. The sample size S needed to ensure a reasonably accurate clustering is constant in the limit, so the some-to-some gather can be done in at worst $O(log(P))$ time for P processes, and the local k-medoids clustering can be performed in $O(S^2)$ time, making the full step worst-case $O(log(P))$. Once local clustering is done, we broadcast medoids from local sampled clusterings to all processes. This is also $O(log(P))$. Each process then chooses its nearest medoid in each clustering in $O(1)$ time, and we use the Bayesian Information Criterion (BIC) [**?**] to select a "best" clustering from the resulting distributed data. The BIC can also be applied in $O(log(P))$ time. Once this is computed, each process assigns its local performance data element to its own nearest cluster (shown in the figure with colors).

Once CAPEK has run, each process knows the cluster of which it is a member, and it has copies of the representatives from all other clusters that enable it to make inferences about the distributed data set as a whole. This setup also enables us to make scalable distributed statistical calculations on each of the clusters, *e.g.* we can compute the mean, variance, max, and min values in logarithmic time.

V. Scalable I/O Metrics

To assess the performance of pF3D at scale, we have developed a custom suite of instrumentation and profiling tools specifically for the interpreter-driven Yorick environment. Specifically, we have developed interception routines to capture and measure the particular POSIX I/O and MPI calls used by Yorick, and we have exported a small number of marker calls that developers can insert into their Yorick code to measure the performance of large-scale I/O dumps.

Using our framework, we measure MPI and I/O bandwidth on all processes at runtime and we record the amount of time spent in POSIX I/O routines and in MPI routines. In addition, we record the transfer speed of these operations and trace the instantaneous bandwidth over time for all processes in a running pF3D application. We take the resulting bandwidth trace and the MPI and I/O profiles and apply our clustering algorithm to the data. We are then able to correlate equivalence classes in the bandwidth trace data with equivalence classes in the profile data, and this allows us to diagnose the root cause of I/O performance problems on-line. Finally, to analyze the performance properties of Yorick's built-in buffering scheme, we record per-process histograms of the bytes written per POSIX write operations. We then correlate particular write size histograms with high or low bandwidth in the parallel pF3D run, which allows us to deduce which buffering strategies work best in the Yorick interpreted environment.

To perform this type of analysis, we developed several data structures, distance metrics, and instrumentation schemes for Yorick. We describe each of these data structures here along with its corresponding distance metric, and we describe how we have input these structures to CAPEK.

f) High-level Yorick Instrumentation: At the highest level, our framework requires support from the application developer to demarcate I/O dumps in the source code. We require this for two reasons. First, our aim was to develop lightweight instrumentation, and we did not want to instrument *all* calls in the MPI library or in the POSIX I/O library. Further, we did not want to incur any instrumentation overhead in regions of pF3D where a dump was not being performed. We require that the pF3D developers insert calls to indicate the beginning and end of their I/O dumps, and we dynamically enable tracing at the beginning of each dump and disable it at the end of the dumps.

g) POSIX I/O and MPI profiles: Within each dump, we use interceptor routines to measure the time spent in each of a subset of POSIX I/O and MPI routines. We record separate profiles for MPI and POSIX I/O so that we can understand the relationship between MPI network traffic and I/O traffic in large systems. To cluster profile data, we use the manhattan distance on vectors of cumulative times in profiles, sorted by the names of the routines being measured. The same type of distance metric is used for MPI and POSIX I/O profiles.

h) Scalable bandwidth traces: In our interceptor routines, we record not only time spent in each routine, but also total bytes transferred by the routine. This enables us to approximate the net MPI and I/O bandwidth achieved by each process. Because these traces may grow to be very long, we use a smoothing technique to normalize a full trace of I/O operations to a fixed-resolution, smoothed bandwidth vector. For our tests, we found that XXX elements was sufficient for accurate problem diagnosis. Again, we used the manhattan distance between these vectors to cluster them across processes.

i) Write size histograms: Finally, as mentioned, we recorded histograms of the sizes of POSIX writes issued by the Yorick interpreter. The number and size of bins was tuned by hand for this work, but we are investigating adaptive techniques for sizing these histograms so that we can use them in a fully generic, production version of our prototype tool. To cluster histograms, we used a method similar to that for our profiles: we applied the manhattan distance to the bin values of histograms.

Using these techniques in conjunction with CAPEK clustering has allowed us to explain I/O dump timings in more detail and to understand the root causes behind the behaviors we saw for various parallel filesystems. We discuss these results and our conclusions in the following sections.

VI. Results

We have run pF3D simulations with a variety of I/O schemes on several systems. We have run on three BlueGene/P systems. dawndev at LLNL is a small system with a Lustre parallel file system and 16 compute nodes per I/O node. Intrepid at ANL has both GPFS and PVFS parallel file systems and 64 compute nodes per I/O node. Dawn at LLNL has a Lustre parallel file system and 128 compute nodes per I/O node. These systems provide our key results. We also ran a few tests on Opteron Infiniband clusters to check performance with more zones per process. The Opterons send their I/O across the IB interconnect to the cluster's I/O nodes. The I/O nodes then send the bytes to the parallel file system.

Figure 2 shows the time spent by each process in collecting its spec data in an in-memory file plus the time until the message to the I/O group leader starts to be received. Each process sends the same amount of data each time, but there is large variability in the time from dump-to-dump. In this case the time is short enough that it does not impact the run time of pF3D. That might change on future systems, so we would like to obtain data that would explain the source of variability. This run used BlueGene aware groups.

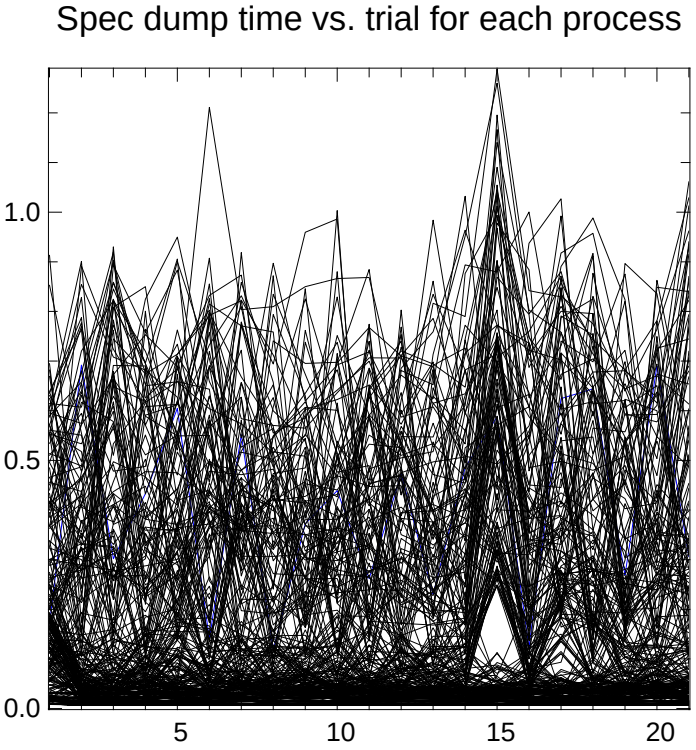

Fig. 2: This figure shows the time spent by each process in collecting spec dump information in a test run on dawndev. The x-axis is the dump number and the times for each process are connected by line segments.

Figure 3 shows a histogram of the time the I/O group leader spends receiving spec data from its group members and writing it out to disk. Yorick's cache block size had no impact on the I/O performance for this run (the black and green curves are basically equivalent). The multi-message dumps had a shorter average time, but the maximum time was similar. The simulation cannot resume running until all data has been written out to disk, so it is the slowest time taht matters. Lustre file buffering was turned on for this run.

Figure 4 is a histogram of the time spent for group members to assemble their visualization data and start sending it to their group leader. All three runs have similar performance. That is reasonable given that this time does not include any writes to disk.

Figure 5 shows the time spent supervising a visualization dump. The times are essentially equivalent for all three runs. The files were identical in all three cases and the time is dominated by disk I/O, so the times should be nearly the same.

Figure 6 shows that the time to write a checkpoint restart file clusters fairly closely for these small (1024 processor) runs. We have observed over a factor of two variability in some large runs, presumably due to the vagaries in the performance of the various Lustre OSTs. A simulation cannot continue until all processes have finished their checkpoints, so having

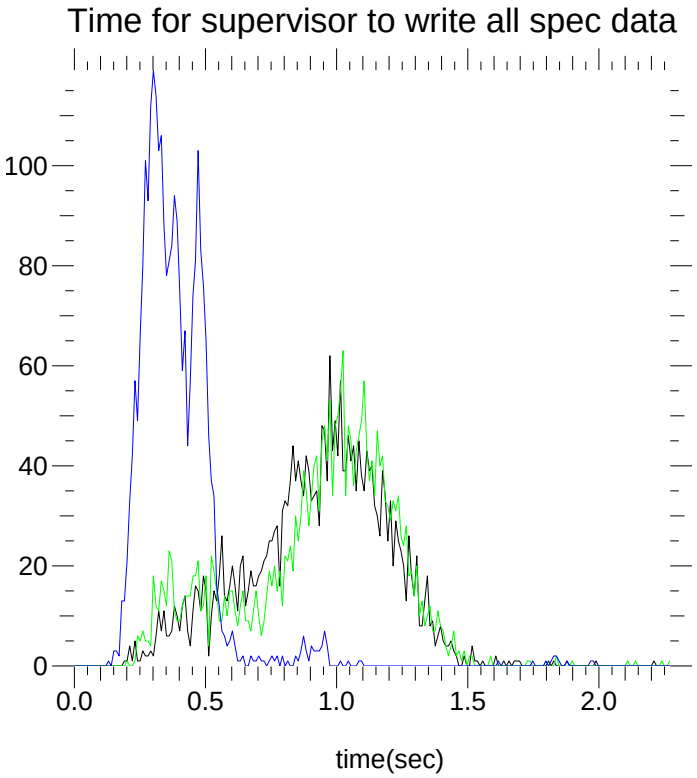

Fig. 3: This figure shows a histogram of the time the I/O group leader spends receiving data from its group members and writing it out to disk for several 1024 process runs on dawndev. The black curve uses a Yorick cache block size of 1 MB and single message dumps. The green curve is the same as the black curve except that it uses Yorick's default cache block size of 64 kB. The blue curve uses a cache block size of 1 MB and multi-message dumps.

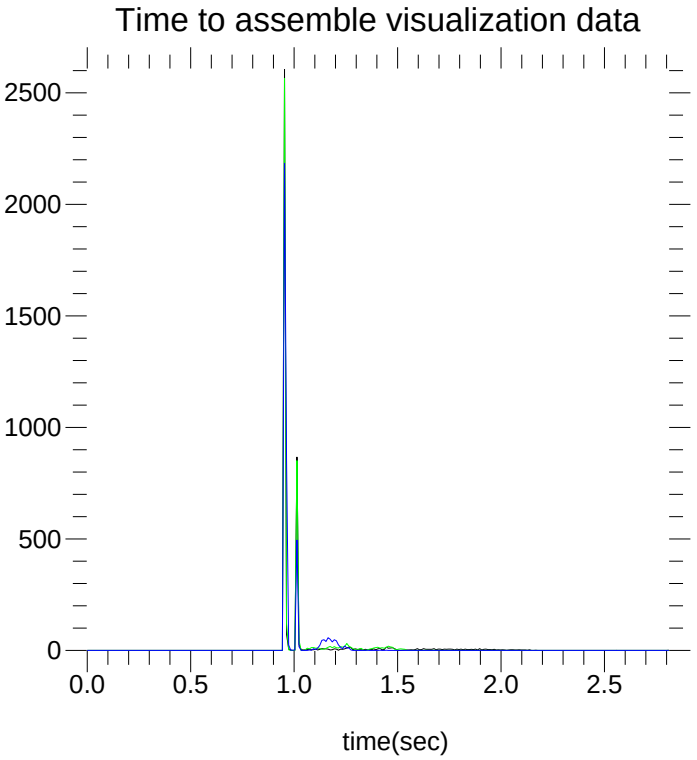

Fig. 4: This figure shows a histogram of the time spent for group members to collect their visualization data and start sending it to their group leader. The colors have the same meaning as in figure 3.

a few stragglers finishing late can greatly reduce effective I/O performance.

j) Hypothesis testing: One of the steps in improving I/O performance is to form hypotheses about how I/O methods interact with the hardware. A sample hypothesis is that short messages hurt the I/O performance of spec dumps. The results presented in the figures show that message size has little, if any, impact on I/O performance in these runs. Achieving good I/O performance at the scale of Sequoia and beyond will require us to make much more sophisticated hypotheses. The detailed data from large runs enabled by the clustering technique will provide us with the information necessary to prove or disprove these hypotheses.

VII. Future Work

The instrumentation described in this paper collects data on the compute nodes. To get a complete picture of I/O performance, we also need instrumentation on I/O nodes (e.g. to assess congestion due to simultaneous access from many clients) and on the computers that are part of the parallel file system (e.g. to determine the OSTs assigned to a particular file by Lustre or to measure the time required for metadata operations). These areas will be a focus for our future work, as well as continuing to work on scalability as we move to tens or hundreds of millions of compute threads.

[**?**]

VIII. Conclusion

The results presented in this paper show that we have the tools necessary to gather performance data for massively parallel runs of pF3D, a code whose I/O is written in the yorick interpreted language. These tools will allow us to investigate different I/O schemes and understand why they do or don't work well on a particular parallel file system and/or a particular supercomputer. We can use these tools on today's petaflop computers to prepare us for the 10-20 petaflop systems arriving in the next couple of years.

Acknowledgment

The authors would like to thank Dave Munro for many helpful discussions about the I/O characteristics of yorick. We would also like to thank ALCF at Argonne National Laboratory for an award of Incite time on the Intrepid BlueGene/P system.

This work was performed under the auspices of the Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344

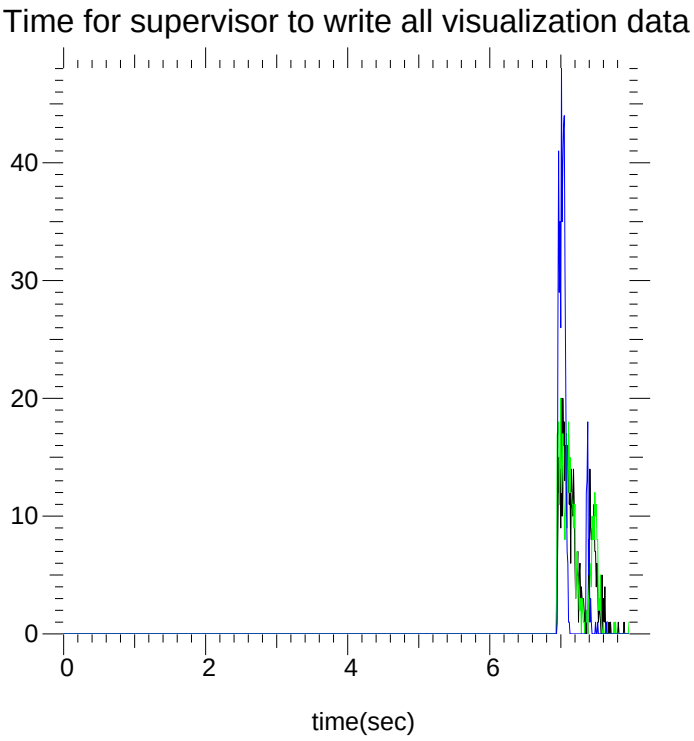

Fig. 5: This figure shows a histogram of the time the I/O group leader spends receiving visualization data and writing it out to disk for several 1024 process runs on dawndev. The colors have the same meaning as in figure 3.

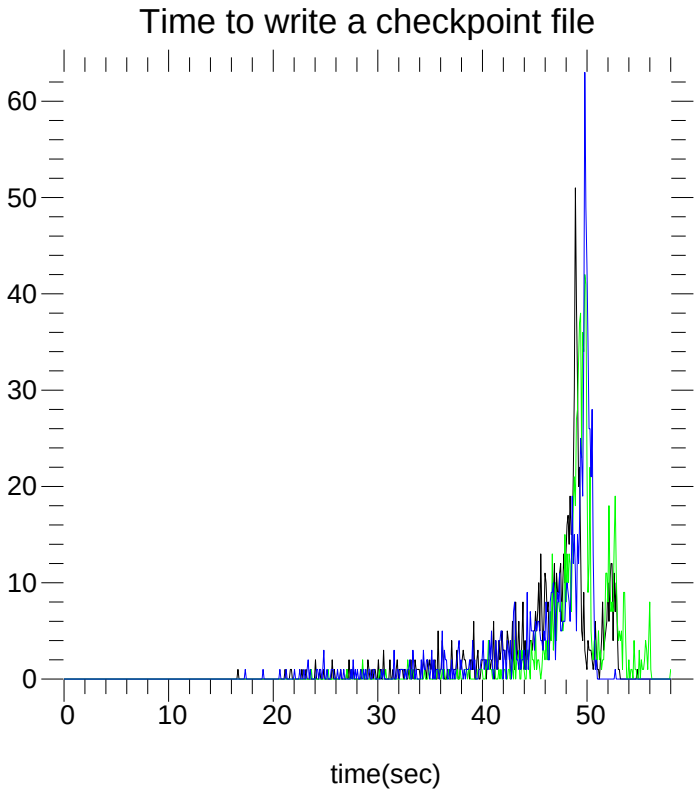

Fig. 6: The figure shows a histogram of the time to write a checkpoint restart file to disk for three schemes. The color code is the same as in previous figures. The dumps are written in file per process mode for all three runs.